# SMP Superscalar (SMPSs) User's Manual

Version 1.0
Barcelona Supercomputing Center
July 2007

# Table of Contents

1Introduction....3
2Installation....3
2.1Compilation requirements....3
2.2Compilation....3
2.3User environment....4
2.4Compiler environment....4
3Programming with SMPSs....4
3.1Task selection....4
3.2SMPSs syntax....5
4Compiling....7
4.1Usage....8
4.2Examples....8
5Setting the environment and executing....9
5.1Setting the number of CPUs and executing....9
6Programming examples....9
6.1Matrix multiply....9
7Internals SMP Superscalar....10
8Advanced features....11
8.1Using Paraver ....11
8.2Configuration file....12
9References....13

# 1 Introduction

This document is the user manual of the SMP Superscalar framework (SMPSs), which is based on a source to source compiler and a runtime library. The supported programming model allows the programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different cores of a multi-core or SMP by means of automatic parallelization at execution time. The requirements we place on the programmer are that the application is composed of coarse grained functions (for example, by applying blocking) and that these functions do not have collateral effects (only local variables and parameters are accessed). These functions are identified by annotations (somehow similar to the OpenMP ones), and the runtime will try to parallelize the execution of the annotated functions (also called tasks).

The source to source compiler separates the annotated functions from the main code and the library calls the annotated code. However, an annotation before a function does not indicate that this is a parallel region (as it does in OpenMP). The annotation just indicates the direction of the parameters (input, output or and inout). To be able to exploit the parallelism, the SMPSs runtime takes this information about the parameters and builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule for execution independent nodes to different cores at the same time. Techniques imported from the computer architecture area like the data dependency analysis, data renaming and data locality exploitation are applied to increase the performance of the application.

While OpenMP explicitly specifies what is parallel and what is not, with SMPSs what is specified are functions whose invocations could be run in parallel, depending on the data dependencies. The runtime will find the data dependencies and will determine, based on them, which functions can be run in parallel with others and which are not. Therefore, SMPSs provides programmers with a more flexible programming model with an adaptive parallelism level depending on the application input data.

# 2 Installation

SMP Superscalar is distributed in source code form and must be compiled and installed before using it.

## *2.1 Compilation requirements*

The SMP Superscalar compilation process requires the following system components:

- GCC 4.0 or later
- GNU make
- Optional: automake, autoconf, libtool, bison, flex

## *2.2 Compilation*

To compile and install SMP Superscalar please follow the following steps:

1. Decompres the source tarball

   ```
   $ tar -xvzf smpss-1.0.tar.gz
   ```

2. Enter into the source directory

   ```
   cd smpss-1.0
   ```

3. Run the configure script, specifying the installation directory as the *prefix* argument. More information can be obtained by running *./configure --help*.

   ```
   ./configure --prefix=/opt/SMPSS
   ```

4. Run make

```
make
```

5. Run make install

```
make install
```

### 2.3 User environment

If SMP Superscalar has not been installed into a system directory, then the user must set the following environment variables:

1. The *PATH* environment variable must contain the *bin* subdirectory of the installation.

```
export PATH=$PATH:/opt/SMPSS/bin
```

2. The *LD_LIBRARY_PATH* environment variable must contain the *lib* subdirectory from the installation.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SMPSS/lib
```

### 2.4 Compiler environment

The SMP Superscalar compiler requires the following programs to be available on the PATH environment variable.

- GNU indent

## 3 Programming with SMPSs

SMP Superscalar applications are based on the parallelization at task level of sequential applications. The tasks (functions or subroutines) selected by the programmer will be executed in the different cores. Furthermore, the runtime detects when tasks are data independent between them and is able to schedule the simultaneous execution of several of them on different cores. All the above mentioned actions (data dependency analysis and scheduling) are performed transparently to the programmer. However, to take benefit of this automation, the computations to be executed in the cores should be of certain granularity (about 80 µsecs or more). A limitation on the tasks is that they should only access their parameters and local variables.

### 3.1 Task selection

In the current version of SMP Superscalar it is a responsibility of the application programmer to select tasks of a certain granularity. For example, blocking is a technique that can be applied to increase the granularity of the tasks in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
void block_addmultiply(double C[BS][BS], double A[BS][BS], double B[BS][BS]) {
int i, j, k;


for (i=0; i < BS; i++)
    for (j=0; j < BS; j++)
        for (k=0; k < BS; k++)
            C[i][j] += A[i][k] * B[k][j];
}
```

## *3.2 SMPSs syntax*

### Starting and finishing SMPSs applications

The following optional pragmas indicate the scope of the program that will use the SMPSs features.

```
#pragma css start
#pragma css finish
```

When the start pragma is reached all the threads are initiated and run until the finish pragma is reached or the program finishes. Annotated functions must be called between this two pragmas when present. If they are not present in the user code, the compiler will automatically insert the start pragma at the beginning of the application and the finish pragma at the end.

### Specifying a task

Notation:

#pragma css **task** *[* **input**(<*input parameters*>) *]*$_{opt}$ *[* **inout**(<*inout parameters*>) *]*$_{opt}$
*[* **output**(<*output parameters*>) *]*$_{opt}$ *[* **highpriority** *]*$_{opt}$
*{ function declaration | function definition }*

Input clause

List of parameters whose input value will be read.

Inout clause

List of parameters that will be read and writen by the task.

Output clause

List of parameters that will be written to.

Highpriority clause

Specifies that the task will be sent for execution earlier than tasks without the highpriority clause.

Parameter notation:

<*parameter*> *[* [<dimension>] *]*$^{*}$

Examples:

In this example, the “factorial” task has a single input parameter “n” and a single output parameter “result”.

```
#pragma css task input(n) output(result)
void factorial(unsigned int n, unsigned int *result) {
        *result = 1;
        for (; n > 1; n--) {
```

```
            *result = *result * n;
        }
    }
```

The next example, has two input vectors “left”, of size “leftSize”, and “right”, of size “rightSize”; and a single output “result” of size “leftSize+rightSize”.

```
#pragma css task  input(left[leftSize], right[rightSize])  output(result[leftSize+rightSize])
void merge(float *left, unsigned int leftSize, float *right, unsigned int rightSize,
        float *result) {
...
}
```

The next example shows another feature. In this case, with the keyword “highpriority” the user is giving hints to the scheduler: the *lu0* tasks will be, when data dependencies allow it, executed before the ones that are not marked as high-priority.

```
#pragma css task highpriority inout(diag)
void lu0(float diag[64][64]){
...
}
```

## Waiting on data

Notation:

#pragma css **wait on**(<*list of expressions*>)

On clause

Comma separated list of expressions corresponding to the addresses that the system will wait for.

In Example 1 the vector “data” is generated by bubblesort. The wait pragma waits for this function to finish before printing the result.

Example 1:

```
#pragma css task inout(data[size]) input(size)
void bubblesort(float *data, unsigned int size) {
        ...
}

void main() {
```

```
        ...
        bubblesort(data, size);
#pragma css wait on(data)
        for (unsigned int i=0; i < size; i++) {
                printf("%f ", data[i]);
        }
}
```

In Example 2, matrix[N][N] is a 2-dimension array of pointers to 2-dimension arrays of floats. Each of this 2-dimension arrays of floats are generated in the application from annotated functions. The pragma waits on the address to each of these blocks before printing the result in a file.

Example 2:

```
void write_matrix (FILE * file, matrix_t matrix)
{
int rows, columns;
int i, j, ii, jj;

fprintf (file, "%d\n %d\n", N * BSIZE, N * BSIZE);
for (i = 0; i < N; i++)
         for (ii = 0; ii < BSIZE; ii++)
        {
                for (j = 0; j < N; j++){
  #pragma css wait on(matrix[i][j])
                        for (jj = 0; jj < BSIZE; jj++)
                                fprintf (file, "%f ", matrix[i][j][ii][jj]);
                }
                fprintf (file, "\n");
        }
}
```

# 4 Compiling

All steps of the SMPSs compiler have been integrated in a single step compilation, called through `mcc` and the corresponding compilation options, which are indicated in the usage section below. The `mcc` compilation process consists in preprocessing the SMPSs pragmas, compiling for the native architecture with the corresponding compiler and linking with the needed libraries (including the SMPSs libraries).

The current version is only able to compile single source code applications. A way of overcoming this limitation is to provide through libraries the code that does not contain annotations and that is not calling to annotated functions.

## 4.1 Usage

The mcc compiler has been designed to mimic the options and behaviour of common C compilers. We also provide a means to send non standart parameters to the platform compiler and linker. The list of supported options is the following:

```
> mcc -help
-Dmacro              Option passed the preprocessors
-g                   Option passed to the native compilers
-g3                  Option passed to the native compilers
-h/-help             Prints this information
-Idir                Option passed to the native preprocessors
-k/-keep             Keep temporary files
-llibrary            Option passed to the native compiler
-Ldir                Option passed to the native compiler
-noincludes          Don't try to regenerate include directives
-O0                  Option passed to the native compilers
-O1                  Option passed to the native compilers
-O2                  Option passed to the native compilers
-O3                  Option passed to the native compilers
-ofile               Sets the name of the output file
-t/-tracing          Enable program tracing
-v/-verbose          Enables some informational messages
-Wc,OPTIONS          Comma separated list of options passed to the native
compiler
-Wl,OPTIONS          Comma separated list of options passed to the native linker
-Wp,OPTIONS          Comma separated list of options passed to the native
preprocessor
```

## 4.2 Examples

```
> mcc -O3 matmul.c -o matmul
```

Compilation of application file `matmul.c` with -O3 optimization level. If there are no compilation errors, the executable file “matmul” is created which can be called from the command line (“`> ./matmul ...`”).

```
> mcc -keep cholesky.c -o cholesky
```

Compilation with -keep option of cholesky.c application. Option -keep will not delete the intermediate files (files generated by the preprocessor, object files, ...). If there are no compilation errors, the executable file “cholesky” is created.

```
> mcc -O2 -t matmul.c -o matmul
```

Compilation with the -t (tracing) feature. When executing “matmul”, a tracefile of the execution of the application will be generated.

```
> mcc -O2 -Wc,-funroll-loops,-ftree-vectorize,-ftree-vectorizer-verbose=3
matmul.c -o matmul
```

The list of flags after the -Wc are passed to the native compiler (for example c99). These options perform automatic vectorization of the code. Note: vectorization seems to not work properly on gcc with -O3.

# 5 Setting the environment and executing

## *5.1 Setting the number of CPUs and executing*

Before executing a SMP Superscalar application, the number of processors to be used in the execution have to be defined. The default value is 2, but it can be set to a different number with the CSS_NUM_CPUS environment variable, for example:

```
> export CSS_NUM_CPUS=8
```

SMP Superscalar applications are started from the command line in the same way as any other application. For example, for the compilation examples of section 4.2, the applications can be started as follows:

```
> ./matmul <pars>
> ./cholesky <pars>
```

# 6 Programming examples

This section presents a programming example for the block matrix multiplication. The code is not complete, but you can find the complete and working code in the directory <install_dir>/share/docs/cellss/examples/, <install_dir> being the installation directory. More examples are provided in this directory also.

## *6.1 Matrix multiply*

This example presents a SMPSs code for a block matrix multiply. The block size is of 64 x 64 floats.

```
#pragma css task input(A, B) inout(C)
static void block_addmultiply(float C[BS][BS], float A[BS][BS], float B[BS][BS]) {
int i, j, k;

for (i=0; i < BS; i++)
        for (j=0; j < BS; j++)
                for (k=0; k < BS; k++)
                        C[i][j] += A[i][k] * B[k][j];
}

int main(int argc, char **argv) {
int i, j, k;

initialize(argc, argv,  A, B, C);
#pragma css start
for (i=0; i < N; i++)
```

```
        for (j=0; j < N; j++)
                for (k=0; k < N; k++)
                        block_addmultiply( C[i][j], A[i][k], B[k][j]);
    ...
    }
    #pragma css finish
```

This main code will run in the main thread while the block_addmultiply calls will be executed in all the threads. It is important to note that the sequential code (including the annotations) can be compiled with the native compiler, obtaining a sequential binary. This is very useful for debugging the algorithms.

# 7 Internals SMP Superscalar

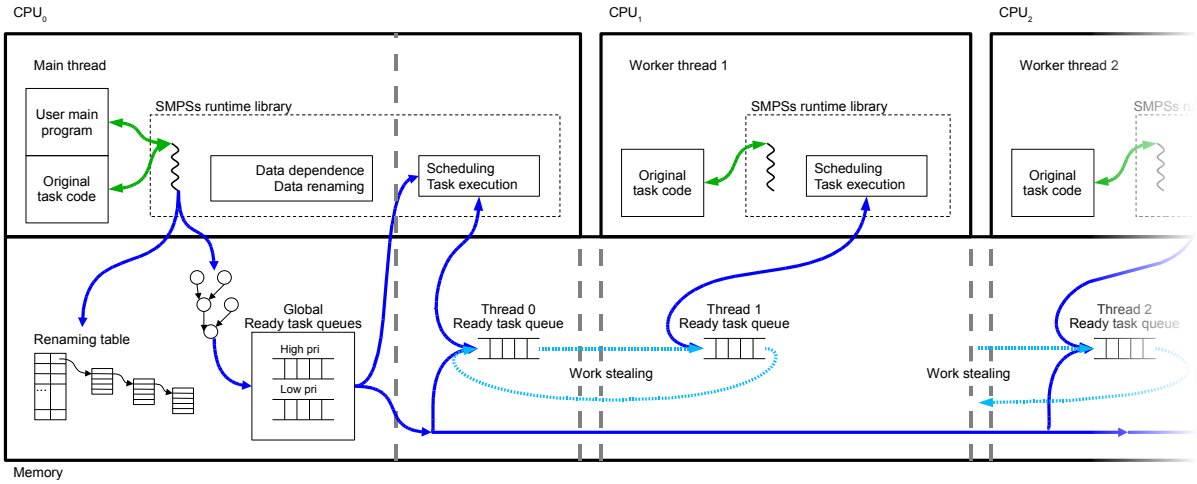


*Figure 1: SMP Superscalar runtime behavior*

When compiling a SMPSs application with mcc the resulting object files are linked with the SMPSs runtime library. Then, when the application is started, the SMPSSs runtime is automatically invoked. The runtime is decoupled in two parts: one runs the main user code and the other runs the tasks.

The most important change in the original user code is that the SMPSs compiler replaces the calls to the css_addTask function whenever a call to an annotated function appears. At runtime, these calls to the css_addTask function will be responsible for the intended behavior of the application. At each call to css_addTask, the main thread will do the following actions:

- A node that represents the called task is added to a task graph.
- Data dependency analysis of the new task with other previously called tasks.
- Parameter renaming: similarly to register renaming, a technique from the superscalar processor area, we do renaming of the output and input/output parameters. For every function call that has a parameter that will be written, instead of writing to the original parameter location, a new memory location will be used, that is, a new instance of that parameter will be created and it will replace the original one, becoming a renaming of the original parameter location. This allows to execute that function call independently from any previous function call that would write or read that parameter. This technique allows to effectively remove some data dependencies by using additional storage, and

thus improving the chances to extract more parallelism.

Every thread has its own ready task queue, including the main thread. There is also a global queue with priority. Whenever a task that has no predecessors is added to the graph, it is also added to the global ready task queue.

The worker threads consume ready tasks from the queues in the following order of preference:

1. High priority tasks from the global queue.
2. Tasks from its their own queue in LIFO order.
3. Tasks from any other thread queue in FIFO order.

Whenever a thread finishes executing a task, it checks what tasks have become ready and adds them to its own queue. This allows the thread to continue exploring the same area of the task graph unless there is a high priority task or that area has become empty.

In order to preserve temporal locality, threads consume tasks of their own queue in LIFO order, which allows them to reuse output parameters to a certain degree. The task stealing policy tries to minimise adverse effects on the cache by stealing in FIFO order, that is, it tries to steal the coldest tasks of the stolen thread.

The main thread purpose is to populate the graph in order to feed tasks to the worker threads. Nevertheless, it may stop generating new tasks for several conditions: too many tasks in the graph, a wait on, a barrier or and end of program. In those situations it follows the same role as the worker threads by consuming the tasks until the blocking condition is no longer valid.

# 8 Advanced features

## *8.1 Using Paraver*

To understand the behavior and performance of the applications, the user can generate Paraver tracefiles of their SMP Superscalar applications.

If the -t/-tracing flag is enabled at compilation time, the application will generate a Paraver tracefile of the execution. The default name for the tracefile is “gss-trace-*id*.prv”. The name can be changed by setting the environment variable CSS_TRACE_FILENAME. For example, if it is set as follows:

```
> export CSS_TRACE_FILENAME=tracefile
```

after the execution, the files: tracefile-0001.row, tracefile-0001.prv and tracefile-0001.pcf are generated. All these files are required by the Paraver tool.

The traces generated by SMP Superscalar can be visualized and analyzed with Paraver. Paraver is distributed independently of SMP Superscalar and can be obtained from:

http://www.cepba.upc.es/paraver/

Several configuration files to visualise and analyse SMP superscalar tracefiles are provided in the SMP Superscalar distribution in the directory <install_dir>/share/cellss/paraver_cfgs/.

The following table summarizes what is shown by each configuration file:

| Configuration file | Feature shown |
|---|---|
| execution_phases.cfg | Profile of percentage of time spent by each thread (master, worker) at each of the major phases in the run time library (ie. generating tasks, scheduling, task execution, ...) |
| flushing.cfg | Intervals (dark blue) where each thread is flushing its local trace buffer to disk. The effect of the flushing overhead on the main thread is of significance since it prevents the main thread from adding newer tasks to the |

| | |
|---|---|
| | graph. This could lead to starvation on the worker threads that would not happen when running without tracing. |
| task.cfg | Outlined function being executed by each SPE |
| task_number.cfg | Number (in order of task generation) of task being executed by each SPE.<br><br>Light green for the initial tasks in program order, blue for the last tasks in program order.<br><br>Intermixed green and blue indicate out of order execution. |
| Task_profile.cfg | Time (microseconds) each thread spent executing the different task.<br><br>Change statistic to:<br>- #burst: number of tasks of each type by thread.<br>- Average burst time: Avg. duration of task type |
| 3dh_duration_tasks.cfg | Histogram of duration of SPE tasks.<br>One plane per task (Fixed Value Selector)<br>Left column: 0 microseconds, right column 3000 ms.<br>Darker: higher number of instances of that duration. |

## *8.2 Configuration file*

With the objective of tuning the behaviour of the SMPSs runtime, a configuration file where some variables are set is introduced. However, we do not recommend to play with this variables unless the user considers that it is required to improve the performance of her/his applications. The current set of variables is the following (values in parenthesis denote the default):

- task_graph.task_count_high_mark(1000): defines the maximum number of non executed tasks that the graph will hold. The purpose of this variable is to control the memory usage.
- task_graph.task_count_low_mark(900): whenever the task graph reaches task_graph.task_count_high_mark tasks, the task graph generation is suspended until the number of non executed tasks goes below task_graph.task_count_low_mark.

These variables are set in a plain text file, with the following syntax:

task_graph.task_count_high_mark = 2000

task_graph.task_count_low_mark = 1500

The file where the variables are set is indicated by setting the CSS_CONFIG_FILE environment variable. For example, if the file “file.cfg” contains the above variable settings, the following command can be used:

```
> export CSS_CONFIG_FILE=file.cfg
```

# 9 References


1. SMP Superscalar website, http://www.bsc.es/plantillaG.php?cat_id=385
2. Josep M. Perez, Rosa M. Badia and Jesus Labarta. ***A flexible and portable programming model for SMP and multi-cores.*** *Technical report 03/2007, Barcelona Supercomuputing Center - Centro Nacional de Supercomputación*, June 2007.
3. Paraver website, www.cepba.upc.edu/paraver